John von Neumann Institute for Computing

Automated Correctness Analysis of MPI Programs with Intel Message Checker

V. Samofalov, V. Krukov, B. Kuhn, S. Zheltov,
A. Konovalov, J. DeSouza

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 901-908, 2006.

http://www.fz-juelich.de/nic-series/volume33

Automated Correctness Analysis of MPI Programs with Intel®Message Checker

Victor Samofalov[a], Victor Krukov[b], Bob Kuhn[c], Sergey Zheltov[c], Alexandr Konovalov[c], Jayant DeSouza[c]

[a]Intel Corporation

[b]Russian Academy of Sciences

[c]Intel Corporation

1. Introduction

Parallel programming is widely considered to be much more complex than sequential programming. When implementing multi-threaded or multi-process parallel algorithms new kinds of errors related to the simultaneous use of shared resources emerge. In addition to introducing new kinds of synchronization errors, the non-determinism of parallel programs can lead to errors that occur intermittently and are hard to reproduce on-demand. Furthermore, old ”sequential” errors can be erroneously interpreted as ”parallel” ones, e.g. writing the wrong value to a global variable is harder to track down in a multi-threaded program. These and other factors significantly complicate implementation and debugging of parallel programs.

For larger parallel systems, distributed parallel programming is the most effective way to improve computing performance. Distributed parallel cluster systems currently dominate high-performance computing, constituting over 70% of the 2004 Top500 list. In most cases, distributed parallel programs use the message-passing programming paradigm, and of the various available candidates, portability and performance have led to MPI as the *de facto* standard for cluster programming. However, the variety and complexity of MPI operations (about 200 in the MPI 1.2 standard) has led to the introduction of new kinds of program errors.

This paper examines the specific area of correctness analysis for MPI programs and describes a new correctness tool developed at Intel’s Advanced Computing Center, called Intel® Message Checker (IMC).[1] Intel® Message Checker is a unique tool for the *automated* analysis of MPI programs. It analyzes trace files and detects several kinds of errors with point-to-point and collective operations such as (a) mismatches in message/buffer sizes, data types, and MPI resources, (b) race conditions, and (c) deadlocks and system-buffer related deadlocks. In addition to the comprehensive analysis engine, IMC also features a graphical user interface with broad functionality for program and data representation. IMC can significantly assist an MPI programmer with distributed program analysis and debugging, and helps them find issues earlier in the debugging process and in less time.

An advantage of *trace-based analysis* is that, if it is sufficiently fast and non-intrusive, it can be used to catch intermittent errors by enabling it for all production runs. Message Checker has low perturbation, which allows it to be used on production runs and the trace-based approach is well-suited to finding subtle problems which surface intermittently on runs.

Fortunately, because the nature of message passing reduces shared data, indeterministic and irreproducible errors are not so common for distributed MPI programs as for shared memory ones. But there are other issues unique to large distributed programs — scalability issues for large systems (how does a user find a problem in 500 processes?) and the need for constant performance optimization. A clear advantage of *automated analysis* is that it scales to extremely large systems better than

current-generation debugging tools (the almighty `printf`).

We note also that automated correctness checking tools (such as IMC) can not only be used for debugging, i.e. assisting the programmer to find an error, but go one step further and actually find the error. As such, this new class of tools, which we refer to as *confidence tools*, can help ensure that a program that provides correct results is really correct. Message Checker is widely used by Intel engineers and has been used to detect issues in several non-trivial MPI applications.

In addition to indicating the error, an advanced correctness checking tool should provide maximum information about the program run to aid the programmer in detecting the cause of the problem. Information only about the stack frame of a failed MPI call, which is what is usually available in debuggers, is not enough for effective bug fixing because the user is unable to answer a non-trivial but very important question — "How did we reach this state in the program?". A call stack provides the current call path, but not the history of calls that have led to the program reaching this state, i.e. it provides depth, but not breadth. Well-known interactive debugging tools for large parallel systems do not make the situation clearer for users. But the *call history* is easily provided by a trace-based post-mortem tool such as IMC. The ideal would be a blend of interactive and trace-based program analysis for MPI correctness.

The rest of this paper is organized as follows. MPI's communication primitives provide opportunities to check correctness on various levels as described in the next section. Our initial performance results are described in Section 4.

2. Taxonomy of Detected Errors by Locality of Analysis

By definition, a programming error is a difference between actual behavior and desired behavior (i.e., specification). From the analysis point of view, errors detected by Intel® Message Checker can be divided into four categories:

1. Errors detection which requires only local MPI function information. e.g. a derived data type with overlapping elements on the receiving side, since such types are valid only for send operations.

2. Errors that can be detected by analyzing only local process information, e.g. initiating a non-blocking point-to-point operation without waiting for or freeing the request.

3. Errors related to multiple processes inside one communicator. There are two sub-classes here:

   (a) Point-to-point errors e.g. incompatibility between data types in a send and its corresponding receive

   (b) Collective operation errors, e.g. different processes specifying different reduce operation in an `MPI_Reduce()`.

4. Error detection which does not require information about inter-process communication but requires information about the states of processes instead. The most serious errors in this category are so-called *potential deadlocks*. In our terminology, a potential deadlock is a deadlock that may become actual if the internal behavior of a function changed, e.g. an `MPI_Send()` for 4-byte message can be either blocking or non-blocking depending on the choice of the MPI implementation.

The current version of Intel® Message Checker supports the semantics of the MPI-1.2 standard. In future, IMC will be extended to support MPI-2.

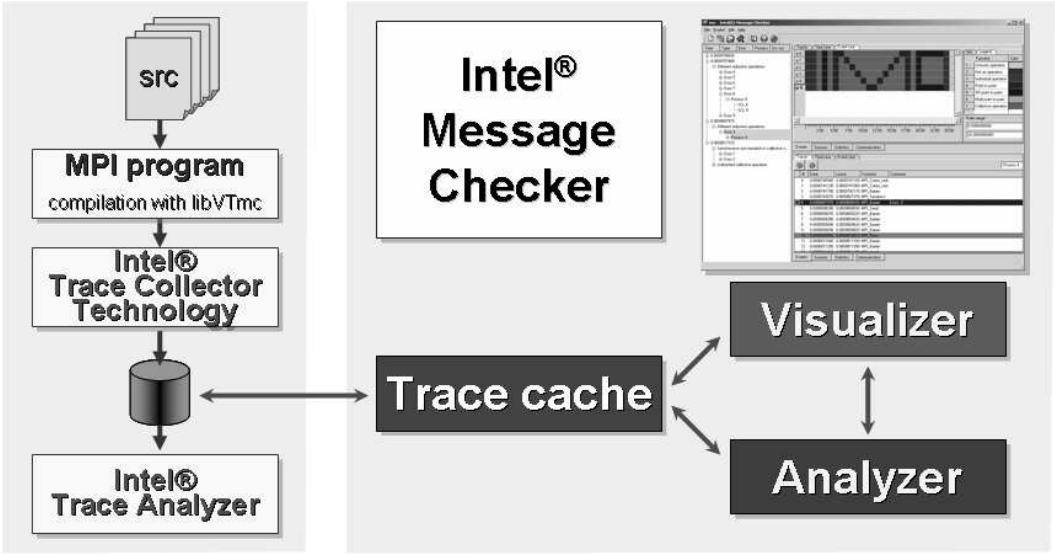

Figure 1. General Intel® Message Checker Software Structure.

As an aside, we note that some error checking can be performed inside the MPI library. An interesting example of such an approach is re-mapping communicators, datatypes, etc. in MPICH2 into integer numbers that are then used for error-checking instead of the original types. This way the MPI library can easily detect invalid arguments (communicators, datatypes, etc.) during an MPI function call.

3. Architecture of Intel® Message Checker and Other Approaches to Implementation

The general software architecture of IMC is presented on Fig. 1. At the linking stage, the MPI program is instrumented with a special version of the Intel® Trace Collector library (the PMPI interface is used inside). Then trace data is collected during the program run and saved into a trace file. Correctness analysis can be performed by either a command-line tool or in the GUI Visualizer environment after the program finishes. The command-line version is targeted for batch/automated testing; the GUI is designated for interactive use in a debugging session.

3.1. Analysis Approach in IMC and Comparison With Related Tools

Other MPI checking tools like Marmot [5] can detect the same set of errors as IMC except deadlocks (the timeout method employed by Marmot cannot guarantee that a real deadlock was detected, and cannot reconstruct the loop in the resource graph which is important for the user to find the real problem). The main difference between Marmot and IMC is in the approach, i.e. Marmot uses an online interactive approach, whereas IMC uses a trace-based offline/post-mortem approach.

In the upcoming MPICH2 release a collective operations checking library [3] will be available but with less functionality then IMC has for such operations.

Both suffice for the “local” error cases described above. But, more complicated error detection requires a more aggressive use of non-local contexts. There are two approaches for this: run-time validity checking or post-mortem trace processing. Most of the existing tools (Marmot [5], collchk [3], NEC run-time MPI checking library [7]) employ run-time analysis.

A significant advantage of run-time checking is the interactive and onine response, i.e. the user

can see the problem report directly as it was found, without spending hours running the program to collect a trace.

But run-time provided error diagnostics may have one disadvantage for the user — they may be not able to see the call history (see Introduction) of the distributed parallel task. Furthermore, the efficient run-time detection of ”distributed” errors and error-prone situations (like deadlocks and race conditions) is quite complex [2]. Inter-process error detection leads to additional communication overhead, which can be quite significant sometimes. These deficiencies are overcome by IMC’s trace-based and post-mortem analysis.

On the other hand, the main disadvantage of the post-mortem approach is the possibility of huge trace files; even modest parallel tasks can generate traces with hundreds of millions of events and of many gigabytes in size. So, tools for post-mortem correctness checking should efficiently support such huge data volumes.

3.2. Visualization Approaches in IMC

In program debugging it is very important for the user to understand how his program appears to be in some state. Saving the history of all program state changes (such as changes to every variable) is unacceptable and also distracts the user with excessive information. One possible solution to this problem is the automatic building of the program’s abstraction model using low-level data and reverse engineering of high-level models. A survey of existing approaches to restoring high-level states can be found in [4]. Such approaches originally were applied to the object-oriented programming model but in general can also be applied to the procedural message-passing paradigm (for example, a call to an object method can be considered as the sending of a message to this object).

Luckily message-passing programming has number of alternative solutions in this area. Visualization methods for message-passing traces have been in development for over 25 years, and are quite mature. They are mostly used in performance analysis, but such a position is too restrictive. Indeed, Gantt chart-based views have many times demonstrated their usefulness during understanding ”what’s going on in the parallel task”.

There are two main views for program structure presentation in IMC Visualizer (see Fig. 2).

1. The Time-Line View — represents actual MPI function calls and the interaction of processes over time. This view reflects the native picture of program execution and may be used to track excessive delays and errors resulting from an unexpected execution order.

2. The Event-Line View — represents the program execution flow and process interaction as a logical sequence of MPI operations. That is, all MPI operations are ordered using some defined relations between operations. For example, all collective operations serializing the execution of multiple processes (`MPI_Barrier`) have the same event number for all processes inside the communicator. This simplifies the analysis of processes’ inconsistent behavior.

4. Performance Results

In the current version of IMC, the user must perform several steps for program correctness checking. First, the user needs to compile their program with a special version of Intel® Trace Collector that is used for correctness data collection. (ITC Message Checker Library (ITCMCL) is a special version of ITC for extended trace data collection used for correctness checking but because of that it is not recommended for accurate performance analysis). Intel® Trace Collector is implemented as overloaded MPI function calls which write data to memory buffers and save them in a background thread to a local temporary file.

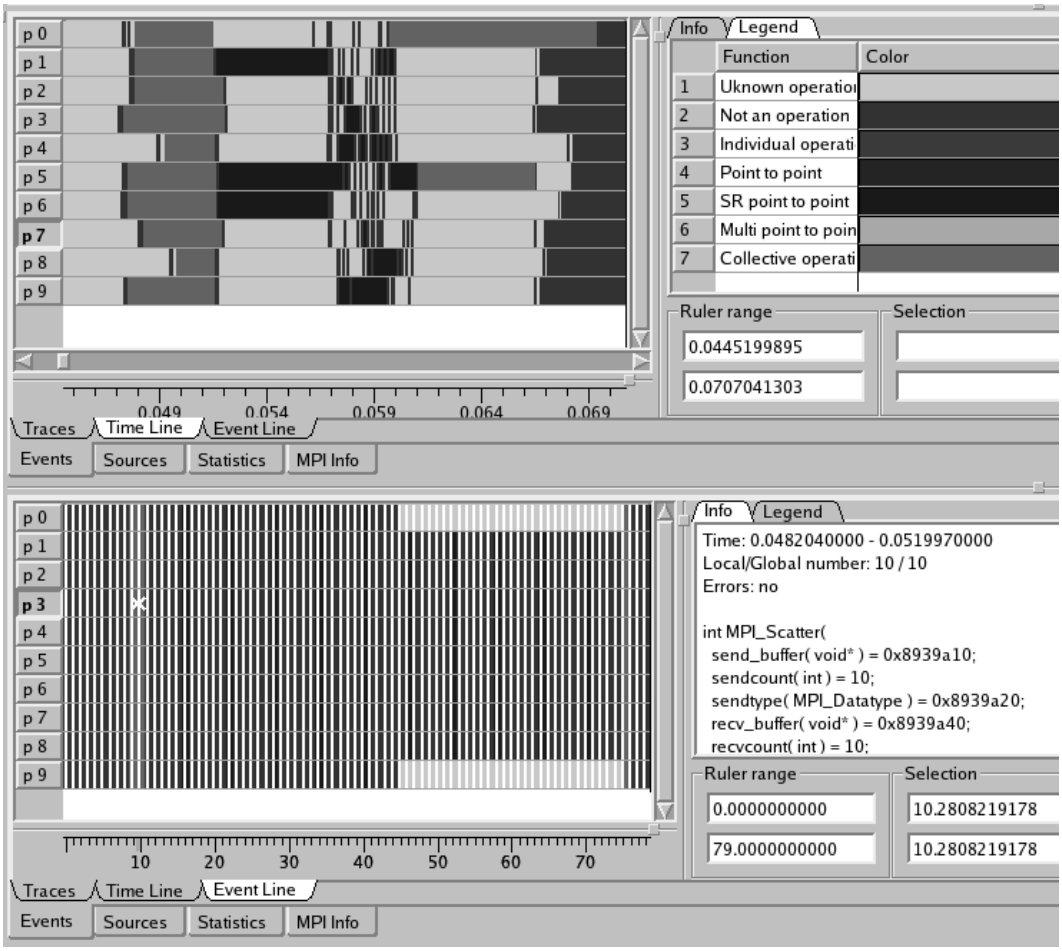

Figure 2. Time Line and Event Line Views.

In Figs. 3–4 we present data for NAS Parallel Benchmarks 2.4 in the Intel MPI Benchmarks(IMB) suite. As can be seen, the overhead of the ITCMCL library is quite small. We should note that standard MPI microbenchmarks like IMB provide a somewhat too optimistic estimation of tracing overhead because they are not aware of ITC trace collection specifics.

Our testbed was a 3-node cluster; each node had four Intel ® Xeon^{TM} 1.4 GHz CPU's with HypherThreading enabled; the interconnect used was Myrinet. The anomalous advantage of 8-process runs over 4-process runs for some benchmarks is caused by a memory bottleneck since the 4-process variant runs on a single node, and 8(9)-process ones run on two nodes).

Some observations from the real-world usage of Intel® Message Checker are:

1. IMC is most usable during program development for debugging and correctness analysis.

2. Even for stable real-word applications, running IMC on ten of them found overlapping of 1-byte receive buffers in two applications (one can be considered a benign fault because the process waited for any event from the other process and did not use the data sent.) This demonstrates the usefulness of confidence tools.

3. IMC is useful even in situation when an error was found by other tools (e.g. MPI itself) because IMC has the program history information for analysis, e.g. when IMC was used on

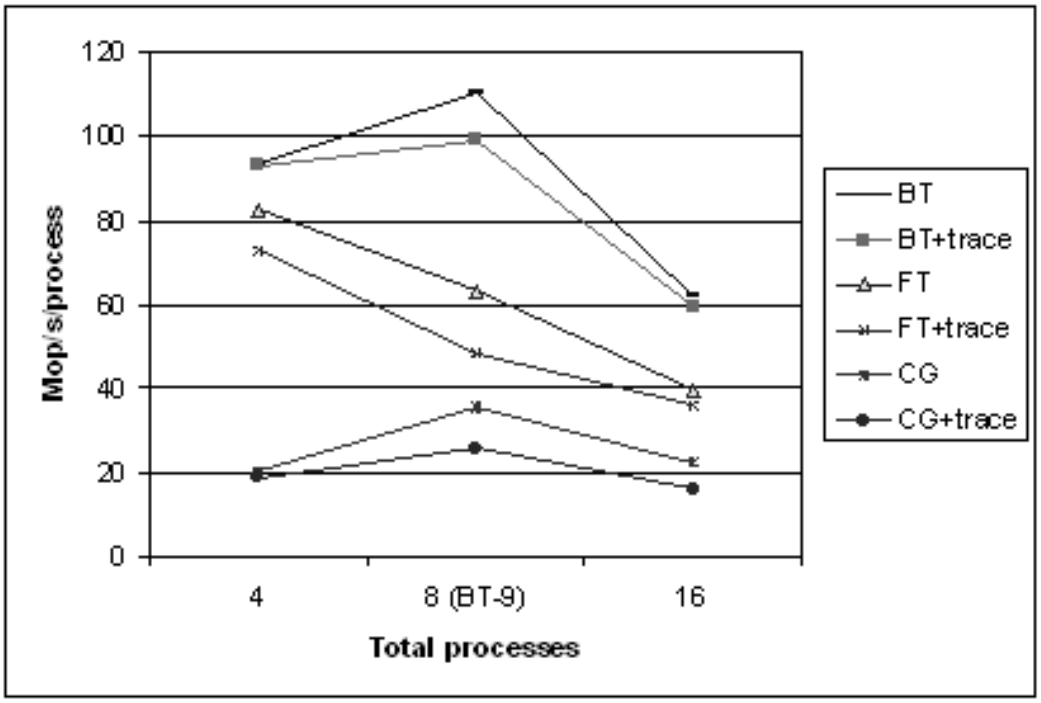

Figure 3. Overhead of correctness checking trace collection: BT, FT, and CG benchmarks.

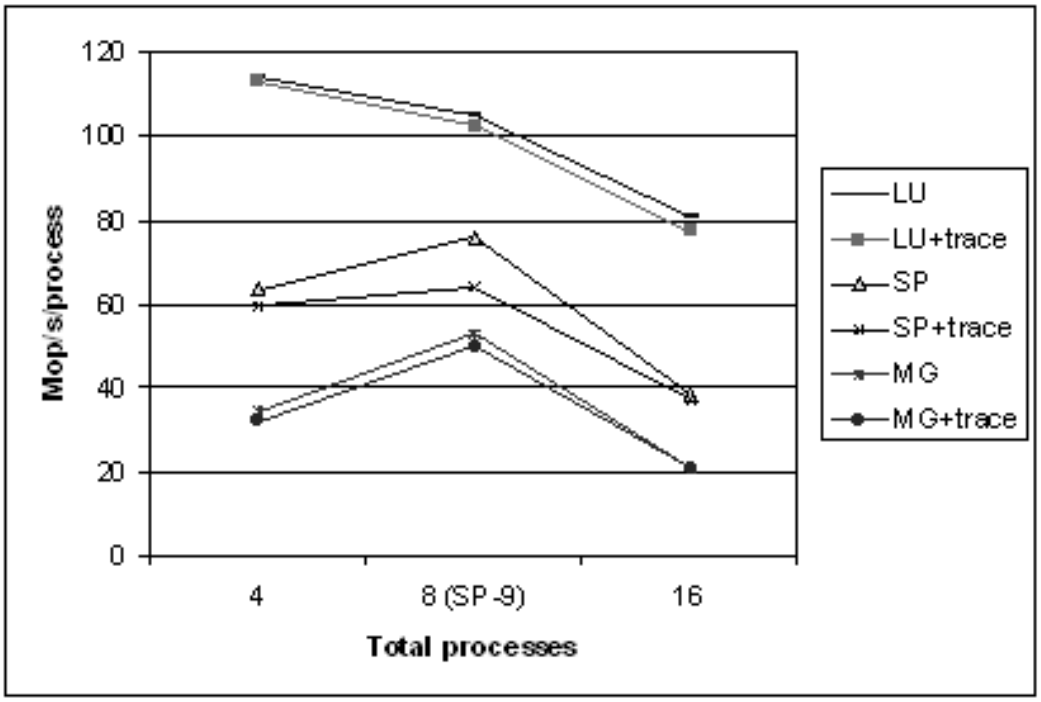

Figure 4. Overhead of correctness checking trace collection: LU, SP, and MG benchmarks.

the GAMESS [6] application — the `MPI_Allgather()` function call with different buffer lengths was first detected by Intel® MPI).

5. Conclusion

We have described the complexity of distributed parallel programming in MPI and motivated the need for a new correctness tool, the Intel® Message Checker from Intel®'s Advanced Computing Center. This tool features a trace-based approach that has low perturbation, high scalability (due to the avoidance of online global analyses), and provides a call history that complements the call stack provided by debuggers. IMC also features an automated analysis that goes futher than a debugger and actually detects errors rather than pointing out symptoms; therefore it has been useful to run it on apparently correct programs. We coined the term *confidence tools* to reflect the new role of such automated correctness tools. The trace-based approach is also useful for catching intermittent errors in large (time or CPU) runs, and the automated analysis provides advantages over manual debugging techniques, especially for large systems. IMC's GUI features an event-line view that separates out logical MPI relationships from the time-centric timeline view.

References

[1] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. // Second International Workshop on Software Engineering for High Performance Computing System Applications, May 2005, St. Louis, Missouri.

[2] R.Cypher, E.Leu. Efficient race detection for message-passing programs with nonblocking sends and receives // Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, 1995.

[3] C.Falzone, A.Chan, E.Lusk, W.Gropp. Collective Error Detection for MPI Collective Operations // Euro PVMMPI'05, 2005.

[4] A.Hamou-Lhadj, T.C.Lethbridge. A survey of trace exploration tools and techniques // Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research Markham, Ontario, Canada, 2004.

[5] B.Krammer, M.S.Müller, M.M.Resch. MPI Application Development Using the Analysis Tool MARMOT // ICCS 2004, Krakow, Poland, June 7-9, 2004. Lecture Notes in Computer Science, Vol. 3038, pp. 464–471, Springer, 2004.

[6] M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, J.A.Montgomery General Atomic and Molecular Electronic Structure System // J. Comput. Chem., 14, 1347-1363(1993).

[7] Jesper Larsson Träff, Joachim Worringen. Verifying Collective MPI Calls // Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings. Lecture Notes in Computer Science, Vol. 3241, pp. 18–27, Springer, 2004.

908